A-013

CurtainRail: 多層の線形構造に基づいた動的空間データ構造
CurtainRail: A linear-structure-based kinetic data structure for spatial searching

多田 瑛貴 *
Tada Teruki

1. はじめに

多次元データを格納し、効率的な索引アルゴリズムを提供する「空間データ構造」は、近年のGIS・機械学習・CG・IoTといった技術の急速な進化により、重要性が一層高まっている。その手法は、Volker Gaede氏とOliver Günther氏が述べているように、すでに多くの種類が発表されている[1]。その中でもR-tree[2]やkd-tree[3]、Quad-tree[4]は既に様々な分野に応用されている上、[1]でも図解されているように、以後発表される多くの新手法の基礎となっている。このような主要な手法の多くは、木構造に基づいている。

しかしながら、木構造に基づいた手法のもとでは、次に述べる2つのタスクにおいて、今以上の顕著な高速化は期待できないと考えられる。

一つは、動的な範囲索引である。ここで範囲索引とは、図*1*のように、特定の多次元範囲(索引範囲)に内包する全データを列挙するクエリを指す。索引範囲が動的に移動・変形する場合において、索引結果は何度も更新される。そのアルゴリズムの高速化は、クラスタリングアルゴリズムの主要な手法の一つであるDBSCAN[5]に見られるように、様々な用途で求められる。しかしながら木構造においては、近隣のデータ同士は、必ずしも構造上近いというわけではない。そのことから、索引結果を更新する際には、基本的に最初から索引をやり直す必要がある。このことから、現状の対数時間を上回る高速性は考えにくい。

そしてもう一つは、データの移動に伴う、データの構造の動的な更新である。木構造に基づく手法においては、同じデータを削除し、位置を変更した上でもう一度再挿入する必要があり、これは非効率である。さらに挿入と削除を繰り返すことで、全体の構造がアンバランスとなり、効率を落とす原因となる。

本研究では、このような木構造の根本的な問題点に対する一つのアプローチとして、多層の線形データ構造に基づいた空間データ構造「CurtainRail」を提案する。この構造では、索引範囲およびデータの移動・変形の際、それぞれの端点の座標の位置関係の入れ替えによって、索引結果を更新する。それにより、変化が小さいほど高速に索引結果を更新できる。十分に小さければ、おおよそ定数時間となると考えられる。また、構造全体の形が変わることもないため、安定した効率が維持される。本研究では、実際にC++を用いてCurtainRailを実装し、既存手法の1つであるR-treeと複数のタスクのもとで計算時間の比較実験を行う。それを踏まえ、CurtainRailの持つ優位性と課題について分析し、今後の展望を考える。

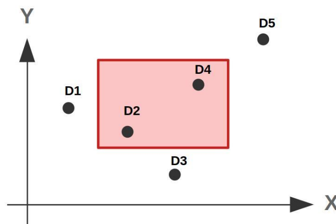

図1 : 空間データ構造による2次元空間での範囲索引のイメージ。
図のDn (n = 1,2,3,4,5) は空間上に分布する点データを表し、中央の枠は索引範囲を表す。この例では、枠内にあるD2とD4が索引の対象となる。

*公立はこだて未来大学 システム情報科学部　Future University Hakodate.

2. CurtainRail

CurtainRailは、主に2種類の線形データ構造で構成されている。一つは、各次元ごとに設けるソート済みの双方向連結リストである。データを格納する際は、各次元の座標の値を、そのデータのアドレスとともにそれぞれのリストに挿入する (なおこの場合、同じデータが次元数分コピーされる。本来であれば、メモリサイズの削減や同期性の確保のため、代わりにポインタを格納するのが望ましい)。そして、座標の値のもとにソートする。もう一つは、索引範囲ごとに設けるハッシュテーブルである。キー(key)は各データのアドレス、値(value)は初期値0の整数フラグとする。この使い方は後述する。

以下より、CurtainRailによる各クエリの流れを示す。なおここでは、扱うデータは全て点データとし、データの数をN、次元数をDとおく。

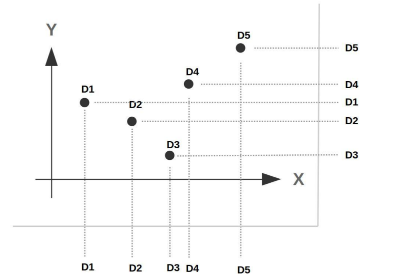

図2 : CurtainRailによる2次元空間でのデータ格納のイメージ

表1 : 図2における双方向連結リストの中身

リストX	D1	D2	D3	D4	D5
リストY	D3	D2	D1	D4	D5

2.1 最初の範囲索引

CurtainRailによる範囲索引の流れは、次のようになる。

(1)索引範囲の各次元の2端点を、データの格納されているものと同じ双方向連結リストに挿入し、ソートする。

(2)各次元で区間索引を行い、結果をハッシュテーブルに記録する。具体的には、索引範囲の2端点間にある全てのデータを取得し、ハッシュテーブルを用いて各データに対応するフラグに1を加算する。これを各次元で行う。すると、ハッシュテーブルには、各データがいくつの次元で区間索引されたかが記録される。

(3)ハッシュテーブルの値をもとに、すべての次元で区間索引されたデータ(=フラグの値と次元数が一致するデータ)を取り出し、結果として出力する。

この一連の流れにより、クエリが満たされる。時間計算量は$O(ND)$である。

Copyright © 2022 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

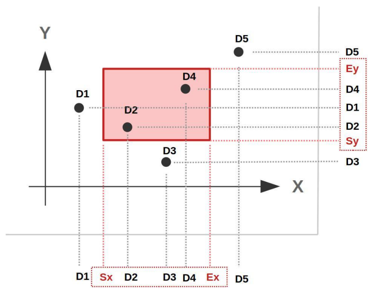

図3： CurtainRailによる範囲索引のイメージ。
データの分布は図2と同様である。中央の枠は索引範囲を表す。

表2：図3におけるリストの中身

リストX	D1	Sx	D2	D3	D4	Ex	D5
リストY	D3	Sy	D2	D1	D4	Ey	D5

表3：図3におけるハッシュテーブルの中身

データ(key)	D1	D2	D3	D4	D5
フラグ(value)	1	**2**	1	**2**	0

2.2 索引結果の更新

範囲索引を1度行えば、ハッシュテーブルの値から随時索引結果を参照することができる。しかしながら、索引範囲が移動・変形した場合、索引結果が途中で変わる場合がある。以下では、その際のハッシュテーブルの更新処理の流れを説明する。

(1)新しいデータの位置をもとに、双方向連結リストを更新する。具体的には、各次元において、移動したデータの座標の値を1つずつ更新し、その都度バブルソートを用いてそのデータの各リストでの順序を更新する。そして、そのデータに対して個別に区間索引を行い、フラグを更新する。それを全データに適用することで、データの移動が構造に適用される。

(2)新しい索引範囲の各次元の2端点を、各双方向連結リストに挿入してソートする。

(3)索引範囲の2端点のうち片側に着目する。古い端点と新しい端点の間にある全てのデータを取得し、ハッシュテーブルから各データのフラグを取り出す。そして、そのデータが新しい索引範囲の区間内に存在するかを調べ、そうである場合はフラグに1を、そうでない場合は-1を加算する。これをもう片側でも行う。最終的には、それをすべての次元に適用する。

(4)古い索引範囲の端点を、全ての双方向連結リストから取り除く。

この一連の流れにより、クエリが満たされる。

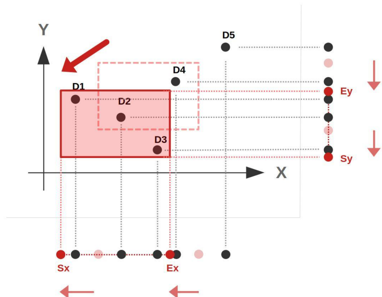

図4：CurtainRailによる索引範囲の移動のイメージ。
データの分布は図2と同様である。

表4：図4におけるリストの中身。(3)で解説したように、索引結果を更新する際は、端点の通る下線部のデータのフラグを更新するのみでよい。

リストX	Sx	<u>D1</u>	D2	D3	Ex	<u>D4</u>	D5
リストY	Sy	<u>D3</u>	D2	D1	Ey	<u>D4</u>	D5

表5：図4におけるハッシュテーブルの中身とその増減

データ(key)	D1	D2	D3	D4	D5
フラグ(value)	**2** (+1)	**2**	**2** (+1)	0 (-2)	0

(1)においては、位置の入れ替わったデータ数をTとすると、時間計算量は$O(TD)$から$O(NTD)$となり、Tの大きさと全体の移動量によって変化する。両方が小さければ小さいほど、計算量は$O(D)$に近づき、おおよそ定数時間となる。(2)(3)においては、時間計算量は$O(D)$から$O(ND)$となり、索引範囲の変化が小さいほど$O(D)$に近づく。

2.3 CurtainRailが優位性をもつパターン

CurtainRailが優位性をもつパターンの条件として考察できるのは、索引範囲の各端点やデータの移動量が、データの分布する全範囲のおよそ$1/N$未満である場合である。そのような状況下においては、一度の更新処理によって入れ変わる各双方向連結リスト内の要素の数が、平均して1個前後となる。その場合、索引範囲の1回の移動またはデータ1つの移動において、実質的に計算量が$O(D)$となる。

この考察については、後述の実験で検証する。

3. パフォーマンスの比較

ここでは、実際にCurtainRailを実装し、主要な既存手法の一つであるR-treeとの計算時間の比較を行う。実装に用いたプログラミング言語はC++で、計算時間の計測にはGoogle社のBenchmark[6]を用いた。なお、実験用のPCは、CPUがIntel(R) Core(TM) i7-11390Hであり、OSはUbuntu 20.04.4 LTS x86_64、主要メモリは16GBである。

3.1 実験A：範囲索引

実験Aでは、範囲索引における2手法のパフォーマンスを比較する。比較に用いる実験は次の条件で行われる。

(1)使用するサンプルのデータ数をN=10000 とする。

(2) 再索引の回数をR = 1000 とする。

(3)サンプルデータは点データであり、1.0x1.0 の範囲中にランダムな位置で生成される。

(4)最初の索引範囲は、データの生成される全範囲(1.0x1.0)の中心にあり、大きさ0.01x0.01である。

3.1.1 実験A-1：索引範囲の変形による索引結果の更新

索引範囲の変形と再索引をR回繰り返す。索引範囲の各端点の1回移動ごとの移動範囲の大きさはMで表される(つまり、各端点の移動量mは、$|m| \leqq M$を満たす)。このとき、CurtainRailの計算量はMの大きさに最も依存すると考えられることから、Mを基準に比較を行うこととする。

本実験では、CurtainRailやR-treeに加え、線形探索での計測も行う。さらに、探索自体を行わず、全データの参照のみを行う場合での計測も行う。

表6：実験A-1の結果．各行のMの値に対応する、
それぞれの手法でのテストの計算時間の計測結果を示す。(単位：ms)

M	CurtainRail	R-tree	線形探索	索引なし
0.01	48,460	1,368	48,517	48
0.001	5,938	995	52,125	49
10^{-4}	1,359	1,418	49,744	48
10^{-5}	637	892	47,776	48
10^{-6}	512	1,660	48,359	49
10^{-7}	470	1,013	48,513	48
10^{-8}	501	1,024	48,654	49

Copyright © 2022 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

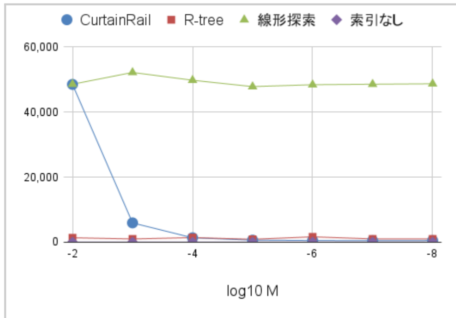

図5 : 実験A-1の結果のグラフ。横軸がMの値の対数、縦軸が計算時間(ms)。

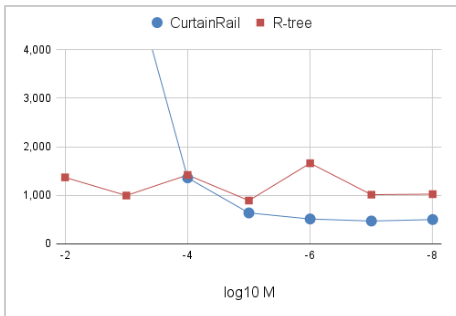

図6 : 図5のうち、CurtainRailとR-treeの結果のみに着目したもの。

結果を表6に示す。この結果を観察すると、CurtainRailによる索引結果の更新は、Mが10^{-5} 未満である場合において、R-treeよりも効率的であることが分かる。一方で、Mがその値より大きく場合においては、計算時間が爆発的に増えるという課題も見受けられる。

3.1.2 実験A-2 : データ1つの移動による索引結果の更新

データ1つの移動と再索引をR回繰り返す。データの1回移動ごとの移動範囲の大きさはMで表される。なお本実験においても、実験A-1と同様の理由により、Mを基準に比較を行うこととする。(以降の実験でも同様である。)

表7 : 実験A-2の結果. (単位 : ms)

M	CurtainRail	R-tree
0.01	1,863	1,777
0.001	791	1,786
10^{-4}	559	1,872
10^{-5}	486	1,686
10^{-6}	501	1,888
10^{-7}	482	1,604
10^{-8}	497	1,643

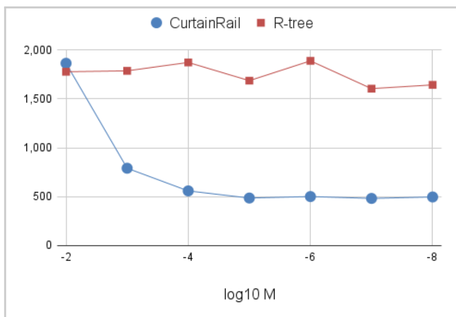

図7: 実験A-2の結果のグラフ。

結果を表7に示す。この結果を観察すると、CurtainRailによる索引結果の更新は、実験対象となっているMの値の範囲において、基本的にR-treeと同等またはより効率的であると分かる。

3.1.3 実験A-3 : 全データの移動による索引結果の更新

全データの移動と再索引をR回繰り返す。Mの定義については、実験A-2と同様である。

表8 : 実験A-3の結果. (単位 : ms)

M	CurtainRail	R-tree
0.01	17,218,336	33,402,586
0.001	9,484,793	25,955,769
10^{-4}	7,914,615	23,821,428
10^{-5}	4,909,281	23,083,391
10^{-6}	4,682,090	22,993,216
10^{-7}	4,789,999	22,880,344
10^{-8}	4,667,848	22,721,169

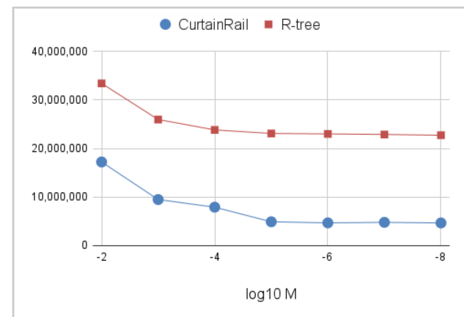

図8: 実験A-3の結果のグラフ。

結果を表8に示す。この結果を観察すると、CurtainRailは実験対象となっているMの値の範囲において、常にR-treeよりも効率的にデータの移動に対応できると分かる。またその範囲内においても、Mが小さくなるほど、R-treeに対する相対的な計算量が小さくなる。

3.1.4 実験A-4 : データ1つの移動と索引範囲の変形による索引結果の更新

データ1つの索引と索引範囲の変形を同時に行った上で、再索引をR回繰り返す。データおよび索引範囲の各端点の、1回移動ごとの移動範囲の大きさはMで表される。
(CurtainRailの用途を考えると、実験A-1や実験A-2に比べ、より現実的なパターンといえる。)

表9 : 実験A-4の結果. (単位 : ms)

M	CurtainRail	R-tree	線形探索	索引なし
0.01	48,460	1,368	48,517	48
0.001	5,938	995	52,125	49
10^{-4}	1,359	1,418	49,744	48
10^{-5}	637	892	47,776	48
10^{-6}	512	1,660	48,359	49
10^{-7}	470	1,013	48,513	48
10^{-8}	501	1,024	48,654	49

Copyright © 2022 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

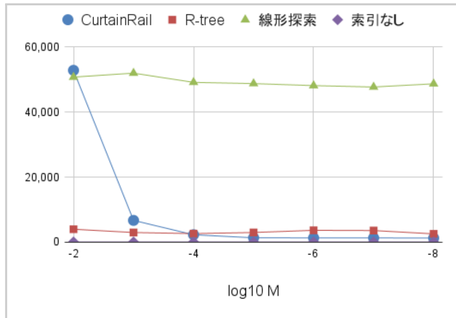

図9: 実験A-4の結果のグラフ。

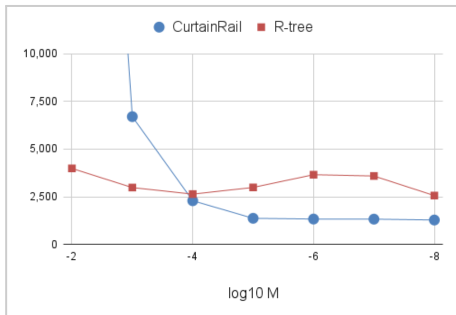

図10: 図9のうち、CurtainRailとR-treeの結果のみに着目したもの。

結果を表*9*に示す。この実験においても実験A-1に同じく、$M = 10^{-5}$ がCurtainrailの優位性の転換点となっている。

以上4つの実験ではいずれにおいても、$M \leqq 1/N$ であれば、CurtainRailの計算時間はR-treeより高速となり、ほぼ一定の値を示す。この結果は、項目2.3で述べたCurtainRailの優位なパターンについての考察にも即していると分かる。

3.2 実験B : DBSCANへの適用

実験Bでは、2手法によるDBSCANのパフォーマンスを比較する。DBSCANの処理においては、索引範囲の移動による索引結果の更新は高頻度に求められる。そのことから、その処理の高速化にCurtainRailが有効ではないかと予想し、実験を行った。比較に用いるテストは次の条件で行われる。

(1)使用するサンプルのデータ数をN=10000 とする。

(2)サンプルデータは点データであり、1.0x1.0 の範囲中にランダムな位置で生成される。ただし、点データの位置は、クラスタリングを適用しやすくするため、特定の領域に集まるような位置をとるように調整される。

(3)DBSCANにおいて、2つのデータを同じクラスに分類する条件となる、データ間の最大距離をd = 0.005とする。

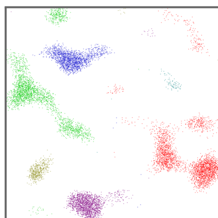
図11 : 実験Bにおける、点データの分布とそのクラスタリングの例。

表10 : 実験Bの結果. (単位 : ms)

CurtainRail	R-tree	線形探索
1,062,963	5,351	1,452,299

結果を表*10*に示す。残念ながら、この実験においては、CurtainRailは線形探索の場合と同等の計算時間を要し、優位性は見られなかった。この原因として考えられるのは、DBSCANの処理において、何度か索引範囲の大きな移動が行われていることが考えられる。

4. 結論

以上の実験結果により、CurtainRailは索引範囲およびデータの移動・変形が十分小さい場合、既存手法よりも高速に索引結果を更新できることが示された。一方で、各要素が大きく移動・変形する場合、計算量が非常に大きくなるという課題も見受けられる。今後の研究の方向性としては、各要素の大きな移動・変形にもある程度対応できるよう、再索引のアルゴリズムを見直す必要がある。

実験結果を踏まえて、現時点で考えられるCurtainRailの実用範囲としては、データの分散範囲に対するデータと索引範囲の変化が小さく、また高頻度な索引情報の更新を要するデータ群を扱う用途が期待できる。その例としては、GISやIoTにおける、位置情報を用いたシステムなどが挙げられる。今後、CurtainRailをどのような分野で活用できるかについても、詳しく調査していきたい。

本研究のもう一つの成果として、線形構造に基づいた動的空間データ構造が、木構造に対して十分な優位性を持ち得ることを明らかにした。現在、木構造以外による空間データ構造の手法の報告例は少ない。そのことから、今後もCurtainRailを筆頭として、線形構造によるより良い手法へのアプローチを開拓していきたい。

謝辞

本研究は、国立情報学研究所主催の「情報科学の達人」プログラムにて行った研究を発端としている。それに際しては、塩川浩昭准教授(筑波大学)をはじめとして、当プログラム関係者の方々から様々な形でご支援や助言を頂いた。また、研究内容や論文の執筆に関して、寺沢 憲吾准教授(公立はこだて未来大学)から多くの有益な助言を頂いた。

参考文献

[1] Volker Gaede and Oliver Günther. 1998. Multidimensional access methods. ACM Comput. Surv. 30, 2 (June 1998), 170–231. https://doi.org/10.1145/280277.280279

[2] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. SIGMOD Rec. 14, 2 (June 1984), 47–57. https://doi.org/10.1145/971697.602266

[3] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. Commun. ACM 18, 9 (Sept.1975),509–517. https://doi.org/10.1145/361002.361007

[4] Finkel, R.A., Bentley, J.L. Quad trees a data structure for retrieval on composite keys. Acta Informatica 4, 1–9 (1974). https://doi.org/10.1007/BF00288933

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96). AAAI Press, 226–231.

[6] Google. 2004. Benchmark. https://github.com/google/benchmark (2022).

Copyright © 2022 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.